Noise Maker

Overview

- Play sounds using the AV Foundation framework.
- Integrate multimedia assets into an Xcode project.

Contents

Part 1	• Discover how to add multimedia assets to an Xcode project.
Part 2	• Add an AVAudioPlayer to the view controller, and import the AV Foundation framework.
Part 3	• Use the AVAudioPlayer API to play a sound file.
Part 4	• Use asynchronous methods to play several sounds simultaneously.
Part 5	• Add a data model to the project to encapsulate all audio playback.
Part 6	• Analyze existing code to determine the frequency of unnecessary object instantiation.
Part 7	• Analyze repetitive code an infer opportunities to use data structures. • Practice using a for-in loop to iterate over an array.
Part 8	• Recognize similarities and differences in repetitive code, and practice extracting varying values into method parameters. • Formulate a logical condition as a Boolean expression.
Part 9	• Apply the tag property to distinguish one interface element from another in code.
Part 10	• Create image sets and add image files to an image set. • Customize the appearance of buttons with images.
Part 11	• Add more buttons and sounds.
Part 12	• Create the launch screen.

Learning Outcomes

- Practice adding interface elements to the main view and establishing Auto Layout constraints.
- Practice connecting buttons to controller actions.
- Discover how to add multimedia assets to an Xcode project.
- Discover the AVAudioPlayer class for playing sounds.
- Discover how application bundles represent the files associated with an app, and how the NSBundle class abstracts the app bundle.
- Observe the Swift forced unwrapping syntax, and evaluate its use in context.
- Practice declaring properties and implementing controller methods.
- Discover the concept of asynchronous methods, and relate asynchronous method calls to how execution can continue while a sound is playing.
- Analyze a code listing and decide what should be encapsulated in a model.
- Distinguish between constants and variables, and between optional and non-optional types.
- Analyze existing code to determine the frequency of unnecessary object instantiation.
- Practice implementing an initializer.
- Analyze repetitive code an infer opportunities to use data structures.
- Describe the difference between mutability and immutability, and relate these concepts to arrays.
- Practice literal array initialization, and accessing array objects with numeric indices.
- Practice using a for-in loop to iterate over an array.
- Recognize similarities and differences in repetitive code, and practice extracting varying values into method parameters.
- Formulate a logical condition as a Boolean expression.
- Practice using the Attributes Inspector and Connections Inspector to modify view attributes and connections.
- Discover the Tag attribute of view elements, and apply the tag property to distinguish one interface element from another in code.
- Discover how Xcode and iOS manage image asset to accommodate different size classes.
- Practice creating image sets and adding image files to an image set.
- Practice customizing the appearance of buttons with images.
- Assess the usability of an interface, and apply accessibility features to accommodate a wide audience of users.

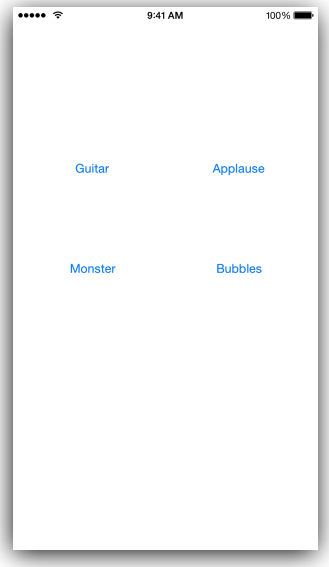
Vocabulary

- Interface Builder
- Object Library
- Button
- controller action
- @IBAction
- .wav file
- Project Navigator
- Xcode Documentation and API Reference
- AVAudioPlayer
- property
- optional
- var
- framework
- AV Foundation
- import
- URL
- path
- NSURL
- app bundle
- NSBundle
- forced unwrapping
- asynchronous method
- Model-View-Controller
- model
- class
- private
- method
- constant
- default value
- instantiation
- initialization
- initializer
- init
- let
- array
- array literal
- mutability
- immutable
- mutable
- for-in loop
- transformation
- map
- closure
- type annotation
- trailing closure
- refactor
- parameter
- argument
- array subscripting
- if
- UIButton
- UIView
- tag property
- Attributes Inspector
- Tag attribute
- Connections Inspector
- asset catalog
- size class
- image set
- resolution
- launch screen
- Auto Layout constraint
- accessibility
- Identity Inspector

Part 1: Add multimedia assets

Objective: Add four buttons to the interface, and add four audio file assets to the project.

1. Create a new project and call it NoiseMaker.
2. Using the Interface Builder Object Library, drag four Buttons onto the main view, and change the labels contents to Guitar, Applause, Monster, and Bubbles.



3. Arrange the buttons into Stack Views.
4. Add constraints to center the main stack view horizontally and vertically.
5. Run the app (⌘ R), and observe the buttons in the interface.
6. Using the Assistant Editor, connect each button to a controller action in the ViewController class.

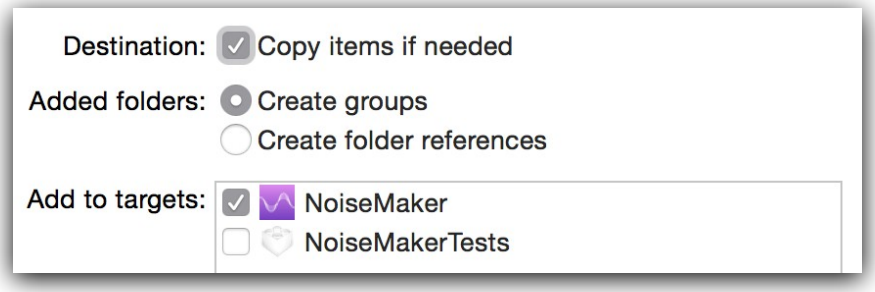
```
@IBAction func playGuitar(sender: UIButton) {
// play guitar sound
}

@IBAction func playApplause(sender: UIButton) {
// play applause sound
}

@IBAction func playMonster(sender: UIButton) {
// play monster sound
}

@IBAction func playBubbles(sender: UIButton) {
// play bubbles sound
}
```

7. Download the Noise Maker Assets.zip file from the materials folder and uncompress it. (Note: the file location changes from year to year. It will be in the class notes on Canvas.)
8. Using the Project Navigator, drag the four audio files guitar.wav, applause.wav, monster.wav, and bubbles.wav to the Supporting Files group.



9. Ensure that Copy items if needed is checked and that NoiseMaker is checked in the Add to Targets: list.
10. To ensure that computer audio volumes are appropriate, click on an audio file in the Project Navigator, experiment with playing the sound within the Xcode interface, and verify that you can hear the sound.

Think about it

- What are .wav files? If we had a lot of big audio files in our app, how might this affect how long it takes to download our app from the App Store, and how much space our app occupies on a device?
- What are other options besides wav format for storing audio files in our app?

Modifications and Extensions

- To maintain organization, add a new group (folder) to the Project Navigator to group the audio files.

Part 2: Add AVAudioPlayer

Objective: Add an AVAudioPlayer property to the view controller, and import the AV Foundation framework.

1. In the View Controller, add a controller property for an AVAudioPlayer that is responsible for playing the guitar sound.

   ```
   var player: AVAudioPlayer?
   ```

2. We declare AVAudioPlayer with an optional type, because the ViewController initializer will not initialize the property with a value.
3. Build the project (⌘ B), and observe the Xcode error notice. In addition to the error, Xcode does not auto-complete nor highlight the AVAudioPlayer class name.
4. Using the Xcode Documentation and API Reference, review the AVAudioPlayer class reference and notice that it resides in the AV Foundation framework.
5. At the top of the ViewController class, add an import statement for the AVFoundation framework.

   ```
   import AVFoundation
   ```

6. Observe that the Xcode editor error notices disappear.
7. Run the app (⌘ R), and observe that Xcode builds and runs the app successfully.

Think about it

- What are the other frameworks that are an essential part of every iOS app?

Part 3: Button Actions

Objective: Implement the actions for each button, playing each of the four sounds.

1. Implement the controller method playGuitar:.

```
@IBAction func playGuitar(sender: UIButton) {
      let url = Bundle.main.url(
            forResource: "guitar", withExtension: "wav")
      if let url = url {
            player = try! AVAudioPlayer(contentsOf: url)
            player?.play()
      }
}
```

2. Run the app (⌘ R), tap the Guitar button, and listen to the guitar sound.
3. URL represents a path to a particular file or network resource.
4. Bundle represents a location of files and resources. The *main* bundle method returns the bundle representing the location of the app files and resources.
5. Implement the playApplause:, playMonster:, and playBubbles: methods. You may be able to copy and paste carefully, changing the file names.

```
@IBAction func playApplause(sender: UIButton) {
      let url = Bundle.main.url(
            forResource: "applause", withExtension: "wav")
      if let url = url {
            player = try! AVAudioPlayer(contentsOf: url)
            player?.play()
      }
}

@IBAction func playMonster(sender: UIButton) {
      let url = Bundle.main.url(
            forResource: "monster", withExtension: "wav")
      if let url = url {
            player = try! AVAudioPlayer(contentsOf: url)
            player?.play()
      }
}
```

```
@IBAction func playBubbles(sender: UIButton) {
      let url = Bundle.main.url(
            forResource: "bubbles", withExtension: "wav")
      if let url = url {
            player = try! AVAudioPlayer(contentsOf: url)
            player?.play()
      }
}
```

6. Run the app (⌘ R), tap on each button, and listen to each sound.
7. Tap on each button quickly, observe how the currently playing sound stops, and how the new sound immediately begins playing.

Think about it

- Why does one sound stop when another begins playing?

Modifications and Extensions

- Bind the four buttons to just one controller method that plays a different sound according to which button is tapped. Note: We are eventually going to consolidate the code through the course of these ten lessons.

Part 4: Play multiple sounds at the same time

Objective: Add four independent AVAudioPlayer properties to the controller, and invoke each player in a respective controller action.

1. In the ViewController class, replace the single controller player property with four distinct AVAudioPlayer? properties.

```
var guitarPlayer: AVAudioPlayer?
var applausePlayer: AVAudioPlayer?
var monsterPlayer: AVAudioPlayer?
var bubblesPlayer: AVAudioPlayer?
```

2. We make each property an optional, since the ViewController initializer will not initialize the properties with values.

3. Update the implementation of each controller action to use each relevant AVAudioPlayer property. Change only two lines in each of the four action methods.

```
@IBAction func playGuitar(sender: UIButton) {
      let url = Bundle.main.url(
            forResource: "guitar", withExtension: "wav")
      if let url = url {
            guitarPlayer = try! AVAudioPlayer(contentsOf: url)
            guitarPlayer?.play()
      }
}
```

4. Explore the AVAudioPlayer play method, and notice the description "play a sound asynchronously." Asynchronous execution means that one method does not have to complete before the next one can start.

5. Run the app (⌘ R), tap each button in quick succession, and listen to multiple sounds playing simultaneously.

Think about it

- What do you think about the repetitive code in our controller? Where is the data model, and what do you think it should it be?

Part 5: Add a data model

Objective: Add a NoiseMaker model to the project to encapsulate all audio playback.

1. Add a new NoiseMaker class (⌘ N) to the project to encapsulate the model.
    a. New → File
    b. iOS → Source
    c. Swift File
    d. Next
    e. Save as "NoiseMaker.swift"
2. Copy the four controller AVAudioPlayer? properties into the NoiseMaker model, mark them as private, and import the AVFoundation framework.

```
import AVFoundation

class NoiseMaker {
      var guitarPlayer: AVAudioPlayer?
      var applausePlayer: AVAudioPlayer?
      var monsterPlayer: AVAudioPlayer?
      var bubblesPlayer: AVAudioPlayer?
}
```

3. Add four new methods to the NoiseMaker class for playing each sound. Copy code that exists in the controller actions into each respective model method.

```
func playGuitarSound() {
   let url = Bundle.main.url(
         forResource: "guitar", withExtension: "wav")
   if let url = url {
         guitarPlayer = try! AVAudioPlayer(contentsOf: url)
         guitarPlayer?.play()
   }
}

func playApplauseSound() {
   let url = Bundle.main.url(
         forResource: "applause", withExtension: "wav")
   if let url = url {
         applausePlayer = try! AVAudioPlayer(contentsOf: url)
         applausePlayer?.play()
   }
```

```
}


func playMonsterSound() {
   let url = Bundle.main.url(
         forResource: "monster", withExtension: "wav")
   if let url = url {
         monsterPlayer = try! AVAudioPlayer(contentsOf: url)
         monsterPlayer?.play()
   }
}

func playBubblesSound() {
   let url = Bundle.main.url(
         forResource: "bubbles", withExtension: "wav")
   if let url = url {
         bubblesPlayer = try! AVAudioPlayer(contentsOf: url)
         bubblesPlayer?.play()
   }
}
```

4. In the ViewController class, remove the the AVFoundation framework import.
5. In the controller, replace the four AVAudioPlayer? properties with a single property for a NoiseMaker model, including a default value.

```
let noiseMaker = NoiseMaker()
```

6. The noiseMaker property is a constant, and, unlike the individual AudioPlayer? properties, is not an optional since a default value is provided.
7. Starting with the playGuitar: method, replace the audio playback contents of each controller action with a method call using the model. You'll have to change all four methods. Each action method will have just the one line in green:

```
@IBAction func playGuitar(sender: UIButton) {
   noiseMaker.playGuitarSound()
}
...
@IBAction func playBubbles(sender: UIButton) {
   noiseMaker.playBubblesSound()
}
```

8. Run the app (⌘ R), tap each button, and observe that the app still plays each sound correctly.
9. Note how the controller now has no concern with the details of playing sounds. The controller merely handles touch events from the view, and communicates with the NoiseMaker model.

Think about it

- We now have a model, but do you feel like we've just "hidden" the repetitive code inside the model? How can we improve this?

Part 6: Initialize the sounds

Objective: Extract the AVAudioPlayer property initialization out of the playback methods and into an initializer.

1. In the NoiseMaker class, extract the AVAudioPlayer instantiations into a new initializer.

```
init() {
   let url = Bundle.main.url(forResource: "guitar", withExtension: "wav")
   if let url = url {
        guitarPlayer = try! AVAudioPlayer(contentsOf: url)
   }

   let url2 = Bundle.main.url(forResource: "applause",
                    withExtension: "wav")
   if let url = url2 {
        applausePlayer = try! AVAudioPlayer(contentsOf: url)
   }

   let url3 = Bundle.main.url(forResource: "monster",
                    withExtension: "wav")
   if let url = url3 {
        monsterPlayer = try! AVAudioPlayer(contentsOf: url)
   }

   let url4 = Bundle.main.url(forResource: "bubbles",
                    withExtension: "wav")
   if let url = url4 {
        bubblesPlayer = try! AVAudioPlayer(contentsOf: url)
   }
}
```

2. Update each "play" method such that they only call the play method on each respective AVAudioPlayer property.

```
func playGuitarSound() {
   guitarPlayer?.play()
}

func playApplauseSound() {
   applausePlayer?.play()
}
```

```
func playMonsterSound() {
   monsterPlayer?.play()
}

func playBubblesSound() {
   bubblesPlayer?.play()
}
```

3. Run the app (⌘ R), and tap the buttons to play each sound.
4. Note how the controller instantiates the NoiseMaker model once, and how the NoiseMaker model instantiates each of its AVAudioPlayer properties only once.
5. Note how tapping each button no longer instantiates a new AVAudioPlayer before playing each sound. The buttons each just play a AVAudioPlayer that has already been instantiated.

Think about it

- How do you think we can reduce the repetitive code in our model?

Modifications and Extensions

- Observe how the app has four buttons, four controller actions, four model methods, and four AVAudioPlayer properties. Investigate how the text property of each button might be used to prepare the AVAudioPlayer properties and to cause the respective AVAudioPlayer object to play the appropriate sound.

Part 7: Make AVAudioPlayer objects

Objective: Replace the individual AVAudioPlayer properties with an array of AVAudioPlayer objects.

1. Add a new [String] property to the NoiseMaker model to store the audio file names. [String] means an array of String objects.

```
class NoiseMaker {

let audioFileNames = ["guitar", "applause", "monster", "bubbles"]
...
```

2. We use array literal syntax to provide a default value for the audioFileNames property, and the let keyword indicates that the array is immutable.
3. Replace the four individual AVAudioPlayer properties with a single [AVAudioPlayer] property to store the AVAudioPlayer objects. [AVAudioPlayer] means an array of AVAudioPlayer objects.

```
var players = [AVAudioPlayer]()
```

4. Using the var keyword indicates that the array is mutable.
5. Within the NoiseMaker initializer, delete the existing repetitive URL and AVAudioPlayer instantiations as follows:
6. Change the implementation of init, using the audioFileNames array and a for-in loop to create new AVAudioPlayer objects. Note how this function is now more compact.

```
init() {
      for filename in audioFileNames {
            let url = Bundle.main.url(
                  for: filename, withExtension: "wav")
            players.append(try! AVAudioPlayer(contentsOfURL: url))
      }
}
```

7. Swift's for-in loop retrieves each String in the audioFileNames array successively, assigning each String to the implicit filename constant during each iteration of the loop.

8. In the body of the for-in loop, for each filename retrieved from the audioFileNames array, a URL is created, and a new AVAudioPlayer object is appended to the players array.
9. Refactor each of the "play" methods to use the players array instead of specific, named AVAudioPlayer properties. Note that you will have to change 4 methods. Change only the green code below.

```
func playGuitarSound() {
      players[0].play()
}
...
func playBubblesSound() {
      players[3].play()
}
```

10. Run the app, and verify that the sounds still play.
11. Note how we have reduced the repetitive code in the model, and decreased the number of lines of code.

Think about it

- We still see repetitive code across both the model and controller "play" methods. Is the repetition related? Can you think of a way we might improve this code even further?

Part 8: Simplify the code

Objective: Refactor the four "play" methods in the model into a single play: method.

1. Note how the play methods in the NoiseMaker model are identical except for the numeric array index.
2. Could we combine the four separate methods into one method that receives an array index as its parameter?
3. In the NoiseMaker class, delete the four independent "play" methods and implement a single play: method.

```
func play(index: Int) {
      players[index].play()
}
```

4. The play: method expects to receive an Int as its argument, which is used to access a particular player in the players array.
5. Note how the NoiseMaker model implementation is now more concise and does not contain repetitive code.
6. However, the implementation of play: is vulnerable to a runtime error if it receives an Int value outside of the bounds of the array, so we should check the value first to ensure safer array subscripting.
7. Update the play: method with increased safety.

```
func play(index: Int) {
   if !players.isEmpty && index >= 0 && index < players.count {
         players[index].play()
   }
}
```

8. We need to make changes in the ViewController too, in order to take advantage of the new NoiseMaker play: method.
9. Update the four controller actions to call the NoiseMaker play: method.

```
@IBAction func playGuitar(sender: UIButton) {
   noiseMaker.play(0)
}
```

```
@IBAction func playApplause(sender: UIButton) {
    noiseMaker.play(1)
}

@IBAction func playMonster(sender: UIButton) {
    noiseMaker.play(2)
}

@IBAction func playBubbles(sender: UIButton) {
    noiseMaker.play(3)
}
```

10. Run the app (⌘ R), tap the buttons, and verify that the functionality remains unchanged.

Think about it

- What do you think about the repetitive code in the controller? Might there be a way to refactor these four methods into one? Can you think of a way to implement this one method without using a long if statement?

Modifications and Extensions

- Add an else clause to the if statement in the NoiseMaker play: method that plays a default sound.
- Extract the condition within the NoiseMaker play: method into a well-named (Int) -> Bool method to encapsulate the checking of the index value. A suitable name for this method might be isValidIndex() or indexIsInRange().

Part 9: Use tag attributes

Objective: Assign values to each button Tag attribute, and bind each button to a single controller action.

1. We could connect the four buttons to one ViewController method that uses an if-else statement to determine which button was tapped. Or, each ViewController action sender argument could be compared to UIButton objects that are ViewController outlet properties. But both these approaches would increase the amount of code, yielding little benefit.
2. Using the Xcode Documentation and API Reference, view the UIButton class reference and notice that it descends from UIView.
3. Using the Xcode Documentation and API Reference, view the UIView class reference and observe the tag property.
4. Using Interface Builder, select a button and view the Attributes Inspector.
5. Note the Tag attribute in the View section of the Attributes Inspector. You can set the Tag attribute to any integer (positive, negative, or zero). All Tags are zero by default.
6. Using the Attributes Inspector, assign each button a Tag value that corresponds to the AVAudioPlayer indices in the model players array property (e.g., Guitar is 0, Applause is 1, Monster is 2, Bubbles is 3).
7. Using the Connections Inspector, delete each button's connection to the respective controller action.
8. Using the Assistant Editor, replace the four controller actions with one new playSound: action. You can type in this code, and we will connect it in the next step.

```
@IBAction func playSound(sender: UIButton) {
    // play the right sound
}
```

9. Using Interface Builder and the Assistant Editor, Control-drag from each button to the single playSound: method to establish action connections.
10. Implement the playSound: method to call the NoiseMaker model play: method, passing the sender parameter tag property value as the argument.

```
@IBAction func playSound(sender: UIButton) {
   noiseMaker.play(sender.tag)
}
```

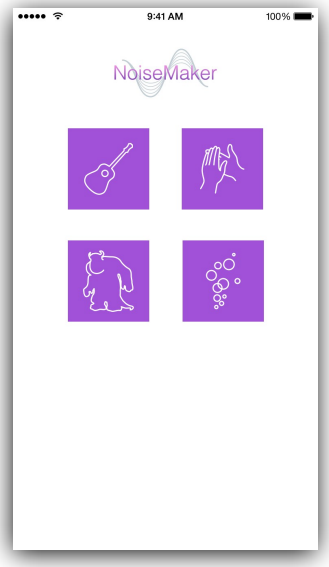
11. The value entered for the Tag attribute in Interface Builder is accessible via the tag property.
12. Run the app (⌘ R), tap the buttons and verify that the sounds still play.
13. Note the significant reduction of code in both the model and controller.

Modifications and Extensions

- Analyze the changes necessary if one were to add an additional button and sound to the app. Add another button and audio file to the project. Describe the subtle dependencies between the user interface and the model, and summarize what code needed to change.
- Consider how the size of the model's AVAudioPlayer array is coupled to the number of buttons and the corresponding tag values. Take a programmatic approach to creating the view, generating buttons depending on the size of the NoiseMaker players array.

Part 10: Add button images

Objective: Add a custom title image to the interface and customize each button's appearance and accessibility attributes.



1. Xcode provides asset catalogs to group and manage assets such as images, that can adapt to different size classes, such as retina screens or different devices.
2. Using the Project Navigator, select the Images.xcassets asset catalog.
3. Download the image files from the location given in class.
4. Drag the 15 supplied image files to the assets area, below the Appicon slot. You will find Applause@2x.png, Applause@2x.png, and Applause@2x.png. You will also find 3 Monster image files, 3 Bubbles image files, 3 Guitar image files, and 3 Title image files.
5. Observe how Xcode displays three different wells for three different sizes of a single image. Different resolutions of devices and size classes might use the different images within an image set. The 1x image might be used on an older device. The 2x image might be used on a device with retina screen. The 3x image might be used on the iPhone 6 Plus.
6. Using Interface Builder, select a button and open the Attributes Inspector. Set the Image attribute to the corresponding image (e.g., Guitar). Within the View panel of the Attributes Inspector, change the Background Color attribute to a desired color. Delete the word in the Title field (e.g., Guitar).
7. Change the size constraints on each button to make them 90 x 90.
8. Repeat the above actions to update each button's appearance, adjust each button position as necessary, and use the menu item Editor > Resolve Auto Layout Issues > Update Constraints to match the constraints to the visible layout on the canvas.
9. Run the app (⌘ R), observe the customized buttons, and tap each button to play a sound.
10. The buttons now lack text, and this might affect accessibility. If the user is blind, their screen reader will not find any text to read out loud.

11. Using Interface Builder, select each button and open the Identity Inspector. In the Accessibility Pane, provide each button a Label (e.g. Guitar), Hint (e.g. Plays guitar sound), and ensure that the Plays Sound Trait is checked.
12. Using the Project Navigator, select Main.storyboard, and use the Object Library to add an Image View to the middle of the top of the interface. Use constraints to make its size 150 x 75, center it horizontally, and place it 20 pixels from the top of the screen.
13. Select the Image View on the canvas, use the Attributes Inspector to set the Image attribute to Title, and observe that the Title image appears on the canvas.
14. Run the app (⌘ R), and observe the image appear in the interface.

Part 11: Add more buttons and sounds

Objective: Add two more buttons and with images and two more sound files.

1. Add the clarinet and harmonica images to the assets catalog.
2. Add two more buttons to the stack views.
3. Using the Attributes Inspector, attach the two new images to the buttons.
4. Add tags 4 and 5 to the new buttons.
5. Add actions for the new buttons.
6. Add the clarinet and harmonica sound files to the project navigator.
7. Add the clarinet and harmonica to the Noise Maker list of sounds.

Part 12: Launch screen

1. Using the Project Navigator, select LaunchScreen.xib, and observe the default app launch screen appear in Interface Builder.
2. Delete the NoiseMaker label, and use the Object Library to add an Image View to the center of the launch screen.
3. Select the Image View on the canvas, use the Attributes Inspector to set the Image attribute to Title, and observe that the Title image appears on the canvas.
4. Use the Align control to center the image view horizontally and vertically, and use the menu item Editor > Resolve Auto Layout Issues > Update Frames to establish appropriate constraints.
5. Run the app (⌘ R), and observe that the Title image appears on the launch screen in the Simulator.